Ciencia de la Computación

Estructuras de Datos Avanzadas

Docente Rosa Yuliana Gabriela Paccotacya Yanque

Boosting K-Means con K-D Tree

Entregado el 29/05/2025

Alexander Rafael Murillo Castillo

Semestre VI

2025-1

"El alumno declara haber realizado el presente trabajo de acuerdo a las normas de la Universidad Católica San Pablo"

_____

Laboratorio 4: Boosting k-Means con K-D Trees

Introducción:

En la era actual de la información, el análisis de grandes volúmenes de datos se ha convertido en una necesidad fundamental para diversas aplicaciones en campos como la inteligencia artificial, el aprendizaje automático, la minería de datos y la visión por computadora (Han, Kamber, & Pei, 2011). Entre las múltiples técnicas de análisis de datos, el clustering destaca por su capacidad para identificar estructuras subyacentes en conjuntos de datos. Dentro de las metodologías de clustering, el algoritmo k-means es una de las más populares debido a su simplicidad y eficiencia (MacQueen, 1967).

El algoritmo k-means busca particionar un conjunto de puntos en k grupos, donde cada punto pertenece al grupo con el centroide más cercano (Lloyd, 1982). Sin embargo, a medida que el tamaño de los conjuntos de datos y el número de clusters aumentan, la eficiencia computacional del algoritmo k-means empeora significativamente. La implementación tradicional de k-means implica cálculos que pueden resultar prohibitivos en términos de tiempo de ejecución para aplicaciones a gran escala (Forgy, 1965).

Para abordar esta limitación, se han propuesto diversas optimizaciones, entre las cuales destaca el uso de estructuras de datos eficientes como el kd-tree. El kd-tree es una estructura de datos que permite realizar búsquedas rápidas de vecinos más cercanos, lo que puede reducir significativamente el tiempo de asignación de puntos a los centroides más cercanos en cada iteración del algoritmo k-means (Bentley, 1975).

El presente trabajo tiene como objetivo implementar y comparar dos variantes del algoritmo k-means: una basada en fuerza bruta y otra optimizada mediante kd-tree. Para ello, se desarrolló un programa en C++ que ejecuta ambas versiones del algoritmo sobre diferentes conjuntos de datos, midiendo los tiempos de ejecución correspondientes. Además, se analizarán los resultados obtenidos para determinar el impacto de usar kd-tree en el rendimiento del algoritmo k-means.

Objetivos:

- Analizar el costo computacional de las implementaciones de k-means basadas en fuerza bruta y kd-tree.
- Ejecutar 10 iteraciones de ambas implementaciones de k-means con un valor fijo de k=18 evaluando las variaciones en los centroides y los tiempos de ejecución, además de visualizar algunos de los resultados obtenidos.
- Evaluar el tiempo de ejecución de ambas versiones de k-means utilizando un conjunto de datos de 2400 puntos con valores de k fijo y n variables.
- Evaluar el tiempo de ejecución de ambas versiones de k-means utilizando un conjunto de datos de 2400 puntos con valores de n fijo y k variables.

Experimentos:

Se llevaron a cabo una serie de experimentos sobre un dataset de 2400 puntos que permiten comparar el rendimiento de las dos implementaciones de k-means en diferentes situaciones. A continuación, se detallan dichos experimentos:

a) **Análisis del costo computacional**

- Evaluar y comparar el costo computacional de las implementaciones de k-means basadas en fuerza bruta y kd-tree. Para ello se evaluarán las implementaciones de ambas versiones de k-means en C++.
    i. **K-means:**
    Empezando por la construcción de los clusters y sus puntos se usa la función:

```
82. unordered_map<Point, vector<Point>> Kmeans::kmeans(int k, int
iterations, bool print) {
 83.     unordered_map<Point, vector<Point>> clusters;
 84.     set<Point> centroids = chooseCentroids(k);
 85.
 86.     if (print) printCentroids(centroids,0);
 87.     for (size_t i = 0; i < iterations; i++) {
 88.         set<Point> newCentroids = recalculateCentroids(centroids,clusters);
 89.         if (print) printCentroids(centroids,i+1);
 90.         if (centroids == newCentroids) break;
 91.         centroids = std::move(newCentroids);
 92.     }
 93.     return clusters;
 94. }
```

Siendo “unordered_map clusters” para almacenar por llave el centroide de cada cluster y como valor se almacena los puntos más cercanos al centroide del cluster.
La función “set<Point> chooseCentroids(int k);” sirve para escoger dentro del conjunto de datos k puntos aleatorios para los centroides de los clusters.

El for “for (size_t i = 0; i < iterations; i++) ” son las iteraciones definidas para calcular la nueva posición de los centroides como la ubicación de los puntos al cluster mediante la función:

```
68. set<Point> Kmeans::recalculateCentroids(set<Point>& centroids,
unordered_map<Point,vector<Point>>& clusters) {
 69.     clusters = clustering(centroids);
 70.     set<Point> newCentroids;
 71.     for (const auto& cluster : clusters)
newCentroids.insert(calcMean(cluster.second));
 72.     return newCentroids;
 73. }
```

Para ello primero ubicamos los puntos a los centroides con la función:

```
 53. unordered_map<Point, vector<Point>> Kmeans::clustering(const
set<Point>& centroids) {
 54.     unordered_map<Point, vector<Point>> clusters;
 55.     for (const auto& point : points) clusters[closest(point,
centroids)].push_back(point);
 56.     return clusters;
 57. }
```

La función “clustering(const set<Point>& centroids)” se encarga de recorrer todos los puntos y cada centroide para calcular el centroide más cercano al punto y almacenarlo en

"unordered_map<Point, vector<Point>> clusters;" para luego retornarlo. Obtenemos un coste **O(n*k*d)** siendo "n" y "d" la cantidad y dimensiones de cada punto.

Una vez obtenido los clusters con sus puntos respectivos se calculará el promedio de todos los puntos de cada cluster para encontrar el nuevo centroide del cluster y retornar el conjunto nuevo de centroides. Teniendo en cuenta el número de d dimensiones se obtiene un coste **O(n*d).**

Habiendo obtenido el conjunto de centroides con sus puntos respectivos y el conjunto de centroides con sus nuevas posiciones evaluamos si los centroides de ambas listas son iguales para romper el bucle "for (size_t i = 0; i < iterations; i++) " se obtiene un coste total:
O(iter*(n*k + n)*d)

ii. **KD-Tree:**
Siguiendo el mismo procedimiento que el k-means solo cambiando la función "clustering(const set<Point>& centroids)" obteniendo:

```
59. unordered_map<Point, vector<Point>> Kmeans::clustering_kd(const
set<Point>& centroids)
60. {
61.     unordered_map<Point, vector<Point>> clusters;
62.     KDTree kdTree(centroids, centroids.begin()->coords.size());
63.     for (const auto& point : points)
clusters[*kdTree.nearestPoint(point)].push_back(point);
64.     return clusters;
65. }
```

En este caso necesitamos construir la estructura KDTree con la inserción de los clusters mediante la función:

```
5. KDTree::KDTree(const std::set<Point>& points, int k) : k(k)
6. {
7.     std::vector<Point> temp(points.begin(), points.end());
8.         root = insert(temp, 0, temp.size(), 0);
9. }
10.
11. KDNode* KDTree::insert(std::vector<Point>& points, int start, int end,  int
depth)
12. {
13.     if (start >= end) return nullptr;
14.
15.     int dim = depth % k, median = (start + end) / 2;
16.     std::nth_element(points.begin() + start, points.begin() + median,
17.                      points.begin() + end,
18.                      [dim](const Point& a, const Point& b) {
19.                          return a.coords[dim] < b.coords[dim];
20.                      });
21.
22.     KDNode* node = new KDNode(k);
23.     node->point = std::move(points[median]);
24.     node->left = insert(points, start, median, depth + 1);
25.     node->right = insert(points, median + 1, end, depth + 1);
26.
```

```
27.     return node;
28. }
```

El cual tiene que hacer un ordenamiento de la lista de clusters por la profundidad de sus ejes, en este caso usamos un ordenamiento de eje con la función STL "std::nth_element" que ubica el punto medio con los puntos menores al punto medio en un lado y los puntos mayores en el otro. Obteniendo un orden promedio de **O(k)** y en el peor de los casos **O(k*log(k))**
Una vez hecho el ordenamiento colocamos las listas de puntos de los extremos del punto medio en " node->left" y " node->right".
Obteniendo un orden total de la función "KDTree(const std::set<Point>& points, int k)" con un promedio de **T(k) = O(k) + 2*T(k/2) = O(k*log(k))** y en el peor de los casos **T(k) = O(k*log(k)) + 2*T(k/2) = O(k*log(k)^2)**

Teniendo los clusters insertados en el KD-Tree podemos hacer las búsquedas de cada punto al cluster más cercano con la función "kdTree.nearestPoint(point)":

```
30. Point* KDTree::nearestPoint(const Point& point)
31. {
32.     double best_dist = std::numeric_limits<double>::max();
33.     return nearestPointRecursive(root, point, 0, nullptr, best_dist);
34. }
35.
36.  Point*  KDTree::nearestPointRecursive(KDNode*  root_,  const  Point&
target, int depth, Point* best, double& best_dist)
37. {
38.     if (!root_) return best;
40.     double dist = distance(root_->point, target);
41.     if (dist < best_dist) { best_dist = dist; best = &root_->point; }
42.
43.     int dim = depth % k;
44.     bool isLeft = target.coords[dim] < root_->point.coords[dim];
45.     KDNode* next = isLeft ? root_->left : root_->right;
46.     KDNode* other = isLeft ? root_->right : root_->left;
47.
48.     best = nearestPointRecursive(next, target, depth + 1, best, best_dist);
49.     if (std::abs(target.coords[dim] - root_->point.coords[dim]) < best_dist) {
50.             best  =  nearestPointRecursive(other,  target,  depth  +  1,  best,
best_dist);
51.     }
53.     return best;
54. }
55.
56. double KDTree::distance(const Point& a, const Point& b)
57. {
58.     return std::sqrt(std::inner_product(a.coords.begin(), a.coords.end(),
59.         b.coords.begin(), 0.0, std::plus<>(),
60.         [](const double& a, const double& b) { return std::pow(a - b, 2); }));
61. }
```

Que se encarga de hacer una búsqueda recursiva desde la raíz hasta las hojas buscando el nodo más cercano mediante la distancia euclidiana de los puntos; para evitar

hacer una búsqueda de todo el árbol se pregunta si la mejor distancia encontrada es menor a la diferencia por la coordenada de profundidad que se evalúa, de modo que impedimos la búsqueda de la rama opuesta si no se cumple la condición. Obteniendo un coste computacional de **O(log(k)*d)** y en el peor de los casos **O(k*d)**

De esta forma k-means con KD-Tree consigue un coste computacional total en el mejor de los casos
O(iter*((n*log(k) + n)*d + k*log(k))), en el peor de los casos
O(iter*((n*log(k) + n)*d + k*log(k)^2)) y en el caso donde la cantidad de k sea muy bajo obtenemos
O(iter*((n*k + n)*d + k*log(k)^2))

Tabla N°1: Comparativa de coste computacional de k-means con k-means con kd-tree

Estructura	**Mejor caso**	**Peor caso**	**Peor caso y k muy bajo**
K-Means	O(iter*(n*k + n)*d)		
K-Means con KD-Tree	O(iter*((n*log(k)+n)*d + k*log(k)))	O(iter*((n*log(k)+n)*d + k*log(k)^2))	O(iter*((n*k+n)*d + k*log(k)^2))

b) **Ejecución de 10 iteraciones con k=18**
- Ejecución de 10 iteraciones de cada implementación de k-means usando k=18 y n= 2400.
- Registro de los tiempos de ejecución y las variaciones de los centroides.
- Visualización de los clusters y centroides usando Python con la librería Matplotlib.

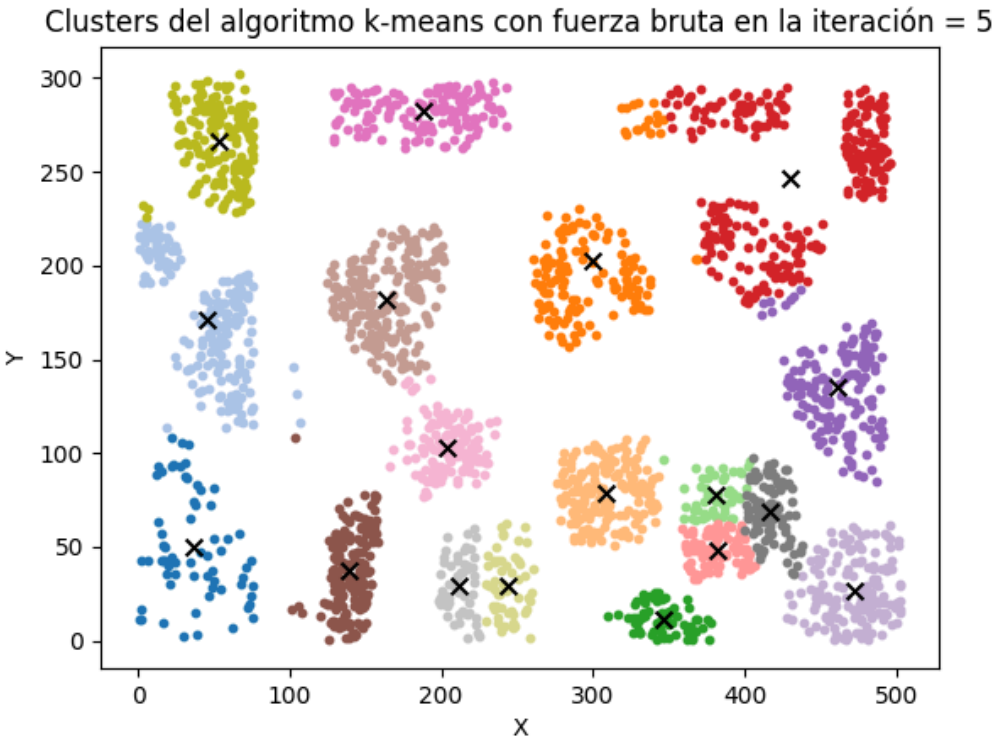
c) Análisis del tiempo de ejecución con k fijo y n variable
- Selección de valores fijos para k = [5,15,25,50,75] y valores variables para n = [1000,1150,1300,1450,1600,1750,1900,2050,2200,2400].
- Ejecución de ambas versiones de k-means para cada combinación de k y n.
- Medición y registro de los tiempos de ejecución.
- Almacenamiento de los resultados y visualización usando Python y Matplotlib.

d) Análisis del tiempo de ejecución con n fijo y k variable
- Selección de valores fijos para n = [1000,1450,1900,2400] y valores variables para k = [5,15,25,50,75,100,125,150,200]
- Ejecución de ambas versiones de k-means para cada combinación de n y k.

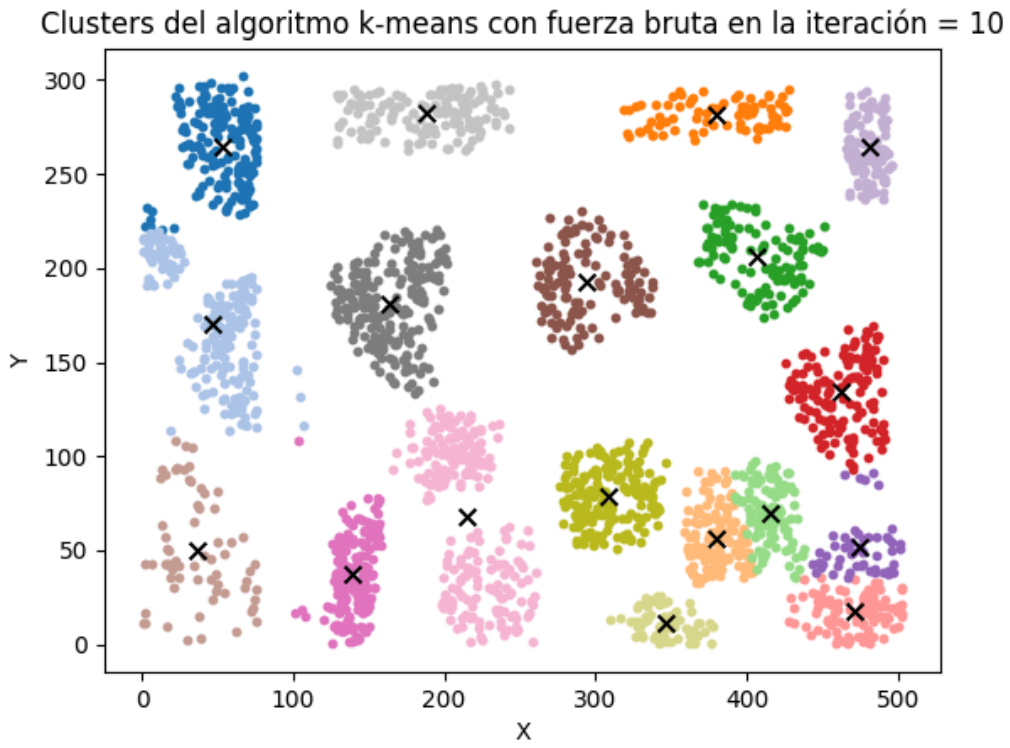
- Medición y registro de los tiempos de ejecución.
- Almacenamiento de los resultados y visualización usando Python y Matplotlib.

Resultados:

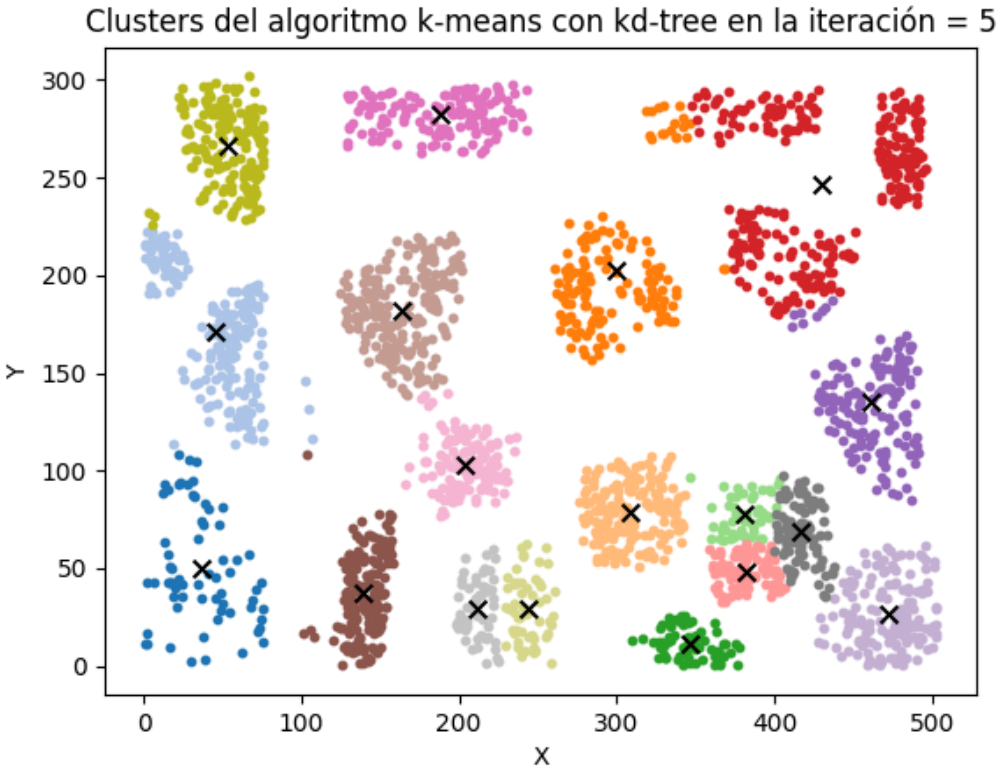
b) Ejecución de 10 iteraciones con k=18



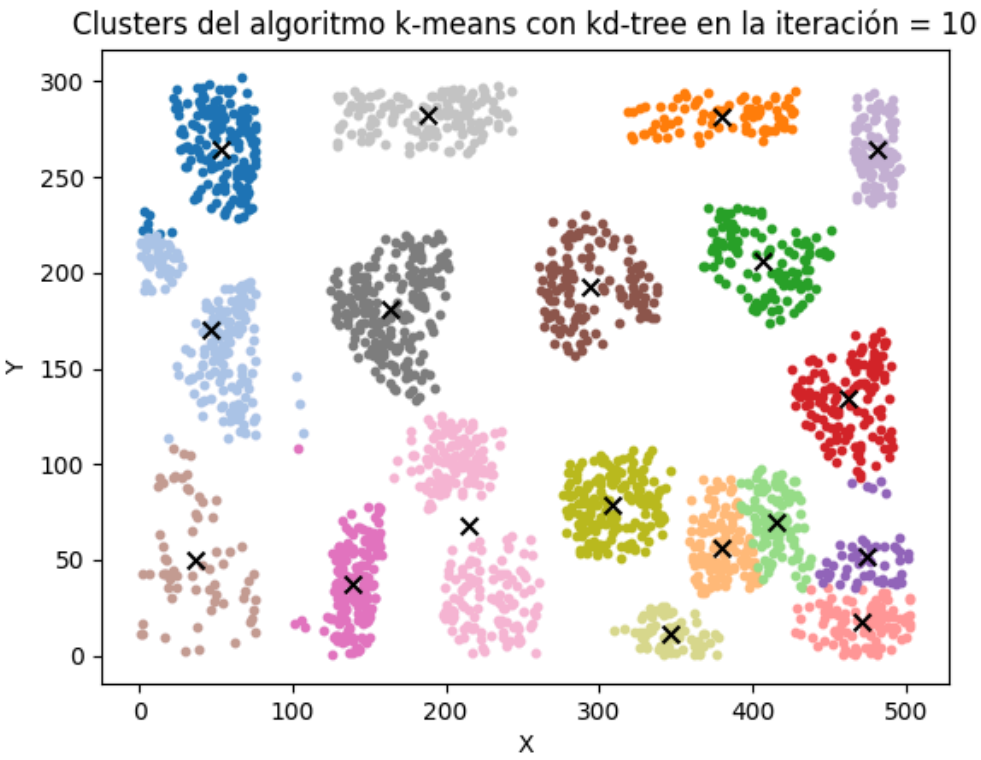
Gráfica N°1: Clusters generados por el algoritmo k-means fuerza bruta(iteración 5)



Gráfica N°2: Clusters generados por el algoritmo k-means fuerza bruta (iteración 10)



Gráfica N°3: Clusters generados por el algoritmo k-means optimizado con kd-tree (iteración 5)



Gráfica N°4: Clusters generados por el algoritmo k-means optimizado con kd-tree (iteración 10)

Tabla N°2: Tiempos de ejecución del algoritmo k-means con fuerza bruta y kd-tree

Iteración	Tiempo de ejecución para K-means fuerza bruta (ms)	Tiempo de ejecución para K-means kd-tree (ms)
1	694.41	311.01
2	631.75	308.74
3	397.50	194.08
4	951.02	460.91
5	692.69	336.32
6	878.07	428.03
7	459.68	221.13
8	459.36	218.78
9	423.71	206.11
10	716.06	350.06
Avg	630.13	303.12

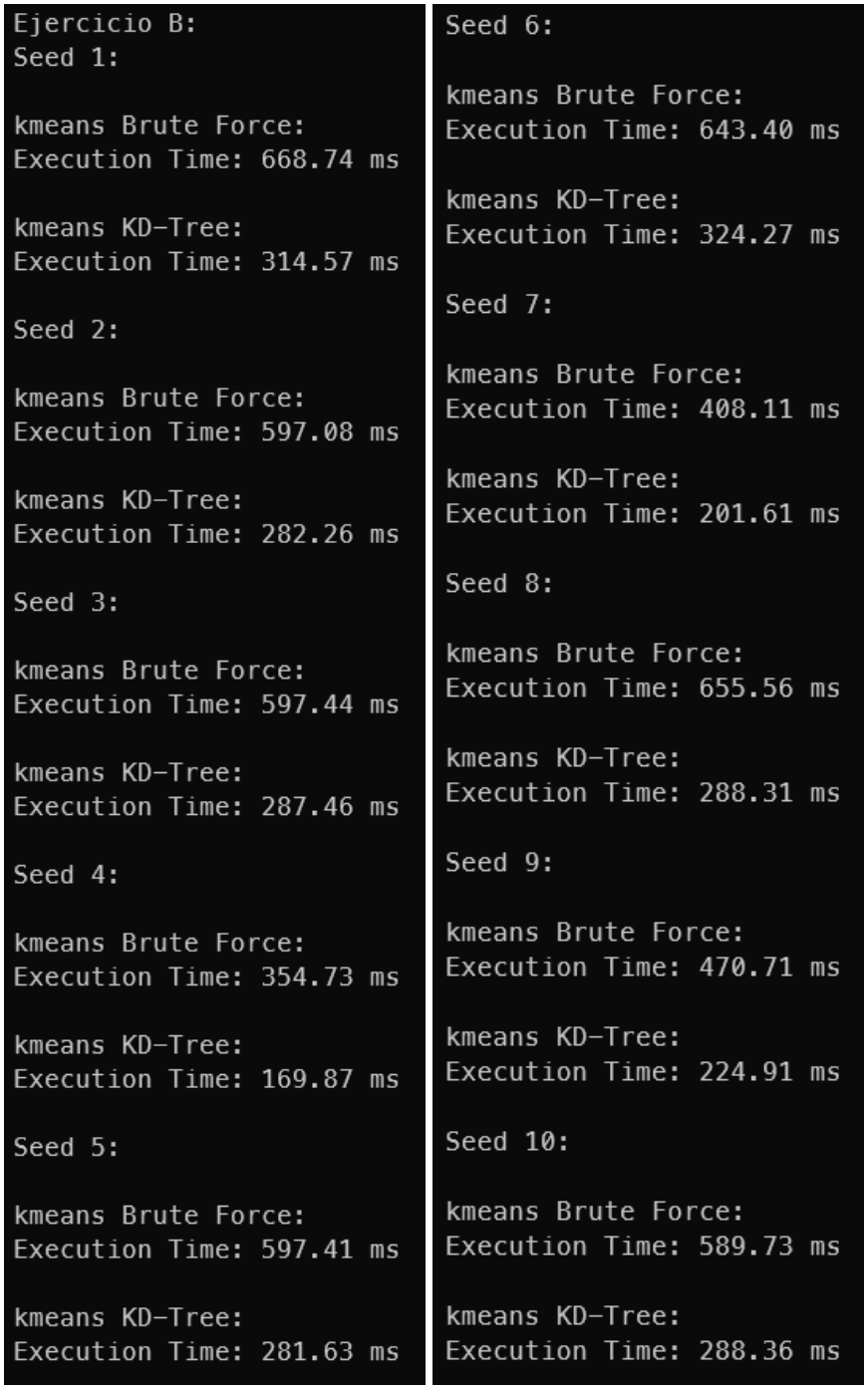
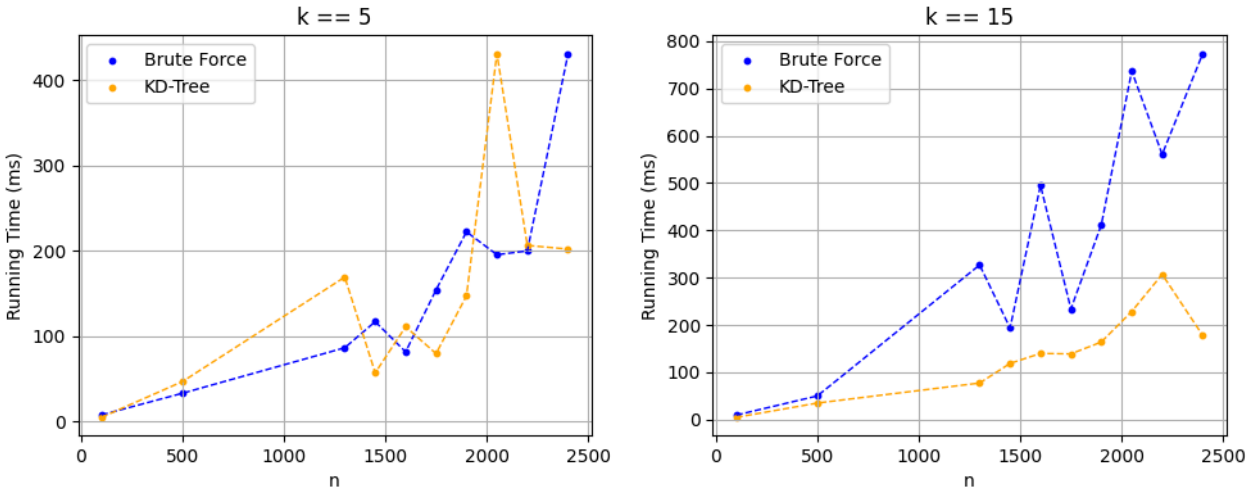


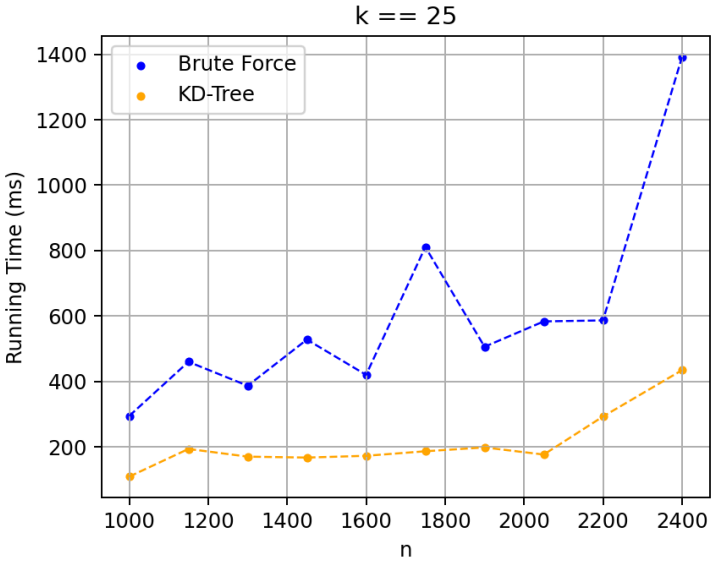
Imagen N°1: Tiempos de ejecución en consola del algoritmo k-means con fuerza bruta y kd-tree

Durante las 10 iteraciones realizadas con k = 18, se observó que ambas implementaciones convergieron hacia soluciones similares en cuanto a la posición de los centroides. Sin embargo, la versión optimizada mediante kd-tree presentó tiempos de ejecución consistentemente mejores.

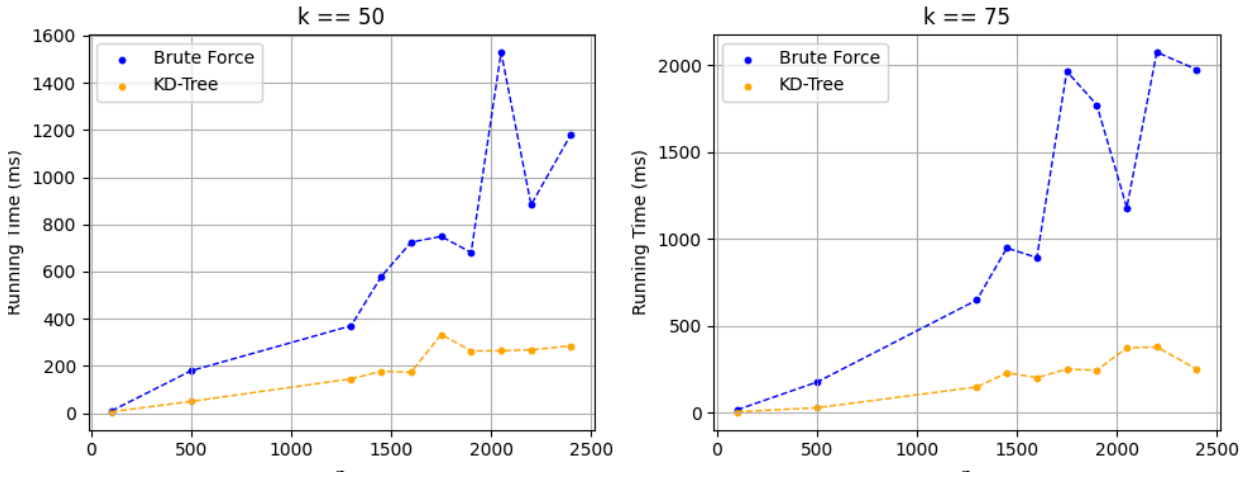
c) Análisis del tiempo de ejecución con k fijo y n variable



Gráfica N°5 y 6: Tiempos de ejecución del algoritmo k-means (fuerza bruta y kd-tree) con k = 5,15 y n variable



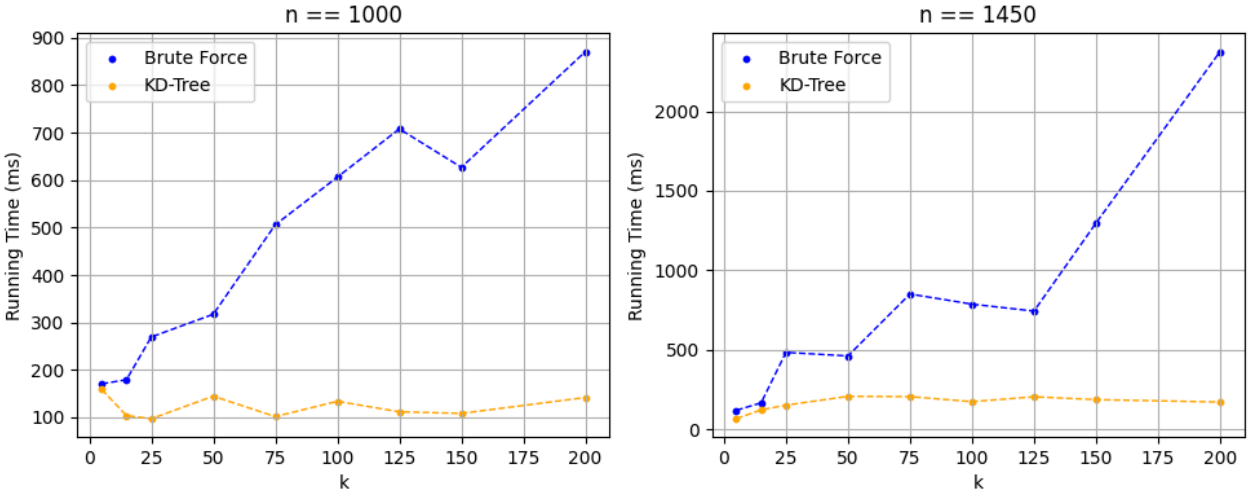
Gráfica N°7: Tiempos de ejecución del algoritmo k-means (fuerza bruta y kd-tree) con k = 25 y n variable



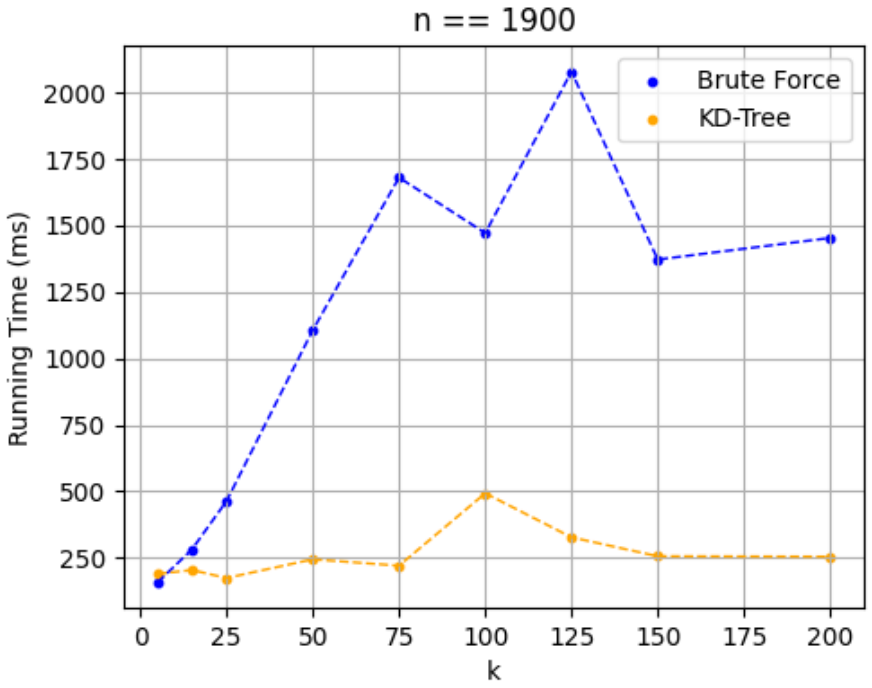
Gráfica N°8 y 9: Tiempos de ejecución del algoritmo k-means (fuerza bruta y kd-tree) con k = 50, 75 y n variable

Se observa que, para un valor pequeño de k el algoritmo de fuerza bruta es más eficiente que el algoritmo con kd-tree para algunos n. Excluyendo esa situación, kd-tree redujo el tiempo de ejecución a comparación de la fuerza bruta.

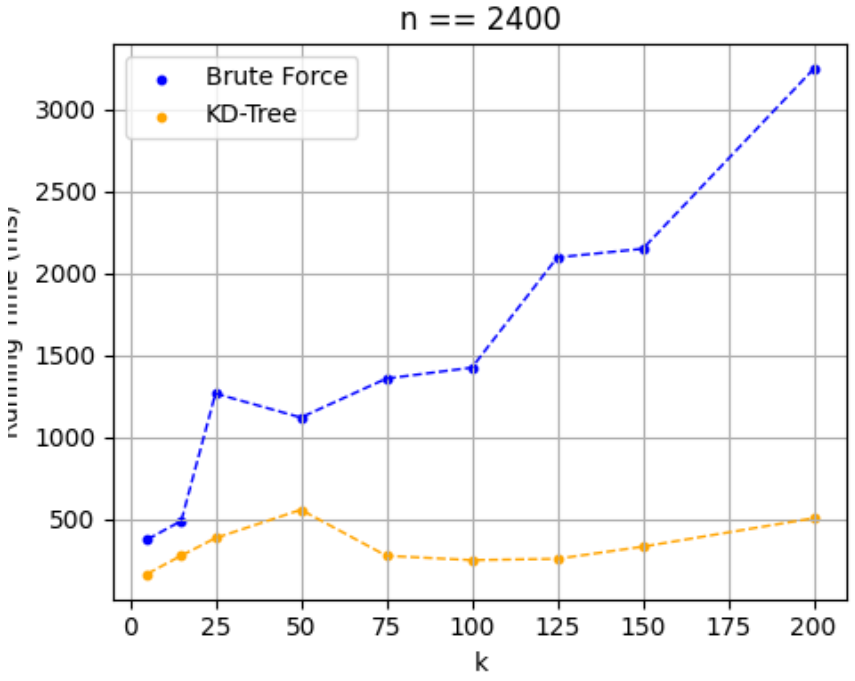
c) Análisis del tiempo de ejecución con n fijo y k variable



Gráfica N°10 y 11: Tiempos de ejecución del algoritmo k-means (fuerza bruta y kd-tree) con n = 100,1450 y k variable



Gráfica N°12: Tiempos de ejecución del algoritmo k-means (fuerza bruta y kd-tree) con n = 1900 y k variable



Gráfica N°13: Tiempos de ejecución del algoritmo k-means (fuerza bruta y kd-tree) con n = 2400 y k variable

Se observa que el algoritmo k-means con kd-tree es consistentemente más eficiente que el implementado con fuerza bruta. A medida que k aumenta, esta diferencia entre tiempos de ejecución se hace más grande.

Discusión:

¿Por qué usar KD-Trees y no otras estructuras? ¿Qué otras estructuras podrían usarse?

La razón por la cual se usa kd-trees es porque tiene la capacidad de optimizar operaciones como la búsqueda del vecino más cercano la cual es esencial para el algoritmo k-means. Esta gran eficiencia la logra gracias a su capacidad para dividir recursivamente el espacio en regiones más pequeñas reduciendo significativamente el número de comparaciones necesarias durante una búsqueda y logrando una complejidad computacional de hasta O(logn) (Baeldung, s.f). Sin embargo, esta gran eficiencia solo se da en espacios de baja a moderada dimensionalidad.

Otras estructuras que podrían utilizarse son:

- **Ball tree:** Es una estructura similar a los kd-trees pero utilizan hiperesferas anidadas para dividir el espacio. Son muy eficientes especialmente en espacios de alta dimensionalidad y con datos altamente estructurados. Limita el número de puntos a ser probados durante la búsqueda del vecino más cercano al utilizar la inequidad triangular. Si el punto más cercano actual se encuentra a una distancia r del punto buscado, cualquier otro punto más cercano al punto de búsqueda debe estar a un radio 2r del punto más cercano actual. (Scikit-learn developers, s.f.)

- **VP-tree:** Es una estructura de datos diseñada para optimizar la búsqueda de vecinos más cercanos especialmente en dimensiones moderadas a altas. Su construcción se basa en la selección recursiva de puntos de referencia llamados “Vantage Points” (VP), cálculo de distancias desde el VP hasta los demás puntos del conjunto y la selección de un umbral para la división del espacio en dos subconjuntos: uno con los puntos cuya distancia al VP sea menor o igual al umbral y otra con los puntos cuya distancia al VP sea mayor que el umbral. Finalmente se repite dicho proceso hasta que se cumpla una condición de parada como un número de puntos en un nodo. Un punto negativo a tomar en cuenta de estos árboles es su alto costo de construcción. (Rakotondrasoa et al., 2023)

¿Para qué tamaño de datos en 2D es beneficioso usar un KD-Tree en el k-means? ¿Por qué?

Es beneficioso utilizar KD-Tree con k-means para un tamaño de datos en 2D medianos o grandes. Esto es porque para conjuntos pequeños (n menor a 30) log(n) es comparable a n y los algoritmos de fuerza bruta pueden llegar a ser más eficientes que las implementaciones con kd-tree. (Scikit-learn developers, s.f.)

¿Para qué valor de k beneficioso usar un KD-Tree en el k-means? ¿Por qué?

Es mejor usar un kd-Tree en k-means para un valor alto de k. La complejidad de asignar los puntos a los clusters con fuerza bruta es de O(k*n) donde k es el número de centroides y n el número de puntos. Usar un kd-tree reduce esa complejidad a O(log(k)*n), El beneficio en cuanto a costo computacional será mayor cuanto mayor sea el valor de k. Por otro lado, para valores bajos de k, el costo de la búsqueda del vecino más cercano usando un kd-tree se aproxima a

una búsqueda lineal pero el costo general del algoritmo k-means empeora debido al costo de construcción del árbol que es de O(k*log(k)^2) (Baeldung, s.f).

Conclusiones:

Al observar los gráficos con clusters “k” fijos y “n” puntos variables , observamos que para “k” igual a 5 el algoritmo de fuerza bruta puede ser más eficiente que la implementación con KD-Tree por su coste de construcción; sin embargo, cuando se aumenta los clusters se aprecia la mejora significativa cuando se implementa KD-Tree, mostrando una menor complejidad de búsqueda al reducir los tiempos de asignación de puntos a clusters.

Cuando se mantiene “n” puntos fijos y “k” clusters variables, se observa que a medida que k incrementa, el tiempo de ejecución del KD-Tree es menor a comparación de fuerza bruta que incrementa linealmente su tiempo de ejecución. Confirmando que KD-Tree es más beneficioso al trabajar con un alto número de clusters al reducir el costo computacional de la búsqueda de vecinos más cercanos.

Finalmente, podemos concluir que el uso de una estructura de datos como el KD-Tree es una solución efectiva para mejorar la eficiencia del algoritmo k-means especialmente para una cantidad grande de datos y un número elevado de clusters al permitir búsquedas rápidas de vecinos más cercanos y disminuyendo la cantidad de comparaciones necesarias entre puntos.

Bibliografía:

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM, 18*(9), 509–517. https://doi.org/10.1145/360825.360844

Forgy, E. W. (1965). Cluster analysis of large applications. *Applied Statistics, 14*(1), 21–27. https://doi.org/10.2307/2346737

Han, J., Kamber, M., & Pei, J. (2011). *Data mining: Concepts and techniques* (3rd ed.). Elsevier.

Lloyd, S. P. (1982). Least squares quantization in PCM. *IEEE Transactions on Information Theory, 28*(2), 129–137. https://doi.org/10.1109/TIT.1982.1057464

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability* (Vol. 1, pp. 281–297). University of California Press.

Baeldung. (s.f.). *KD-Trees: An introduction*. Recuperado de https://www.baeldung.com/cs/k-d-trees

Scikit-learn developers. (s.f.). *neighbors module*. En *Documentación de scikit-learn*. Recuperado el 1 de noviembre de 2024, de https://qu4nt.github.io/sklearn-doc-es/modules/neighbors.html

Rakotondrasoa, H. M., Bucher, M., & Sinayskiy, I. (2023). Quantitative Comparison of Nearest Neighbor Search Algorithms. arXiv (Cornell University). https://doi.org/10.48550/arxiv.2307.05235